Analyzing the potential of GPGPUs for real-time explicit finite element analysis of soft tissue deformation using CUDA

Vukasin Strbac[a], Jos Vander Sloten[a], Nele Famaey[a]

[a]*Biomechanics Section, Department of Mechanical Engineering, KU Leuven, Celestijnenlaan 300C, 3001-Heverlee, Belgium.*

Abstract

As the presence of finite element implementations on General Purpose Graphics Processing Units (GPGPUs) is the literature increases, detailed and in-breadth testing of the hardware is somewhat lacking. We present an implementation and detailed analysis of an FE algorithm designed for real-time solution, particularly aimed at elasticity problems applied to soft tissue deformation.

An efficient parallel implementation of Total Lagrangian Explicit Dynamics implementation is elucidated and the potential for real-time execution is examined. It is shown that in conjunction with modern computing architectures, solution times can be significantly reduced, depending on the solution strategy.

The usability of the method is investigated by conducting a broad assay on ranging model sizes and different cards and comparing to an industry-proven FE code Abaqus. In doing so, we study the effect of using single/double precision computation, quantify and present error measurements as a function of the number of time-steps. We also examine the usage of a special texture memory space and its effect on computation for different devices. Adding material complexity in the form of a tissue damage model is presented and its computational impact elucidated. The aggregate results show that, for a particular set of problems, it is possible to compute a simple set of test cases

*Corresponding author: +3216372759/+32474450327; Celestijnenlaan 300C, 3001-Heverlee, Belgium.

Email address: vule.strbac@mech.kuleuven.be (Vukasin Strbac)

30-250 times faster than current commercial solutions.

According to the speedups achieved, an indication is provided that the GPGPU technology shows promise in the undertaking of real-time FE computation.

---

1. Introduction

1.1. Real-time finite element analysis of soft tissues

As the Finite Element method is becoming more pervasive in scientific and engineering endeavors, the application space of the method increases as well. Likewise, computational architectures are changing and computational power advances rapidly. It is therefore prudent to re-examine the performance of existing algorithms on novel hardware and implementations. We present an analysis of an efficient finite element implementation on modern General Purpose Graphics Processing Units (GPGPUs). The potential of real-time execution is examined, particularly aimed at elasticity problems of soft tissue deformation.

Surgery is shifting towards less invasive techniques, which come at the cost of increased complexity, and in the case of telesurgery, a lack of haptic feedback. The accompanying problem of the lack of force-feedback can be partly compensated by intelligent information flow to the surgeon, in the form of mimicking the known elastic behavior of soft tissues. Likewise, virtual reality training simulators benefit from increased visual and haptic fidelity in the treatment of tissue deformation. It is clear that in both use cases execution time is critical.

In essence, a biomechanical simulation of the surgical situation is required. As with virtual surgical simulators, real-time requirements solicit efficient computation schemes capable of dealing with large deformation of soft biological tissue, many of which are known to behave in a nonlinear hyperelastic fashion. Implementations of various algorithms on different novel many-core GPGPU devices have shown promise in fulfilling these speed constraints. However, the rapid pace of advancement in computational technology necessitates the reimplementation and reevaluation of existing solutions on novel hardware. With the evident rate of increase in their performance and complexity, this is especially true for many-core GPGPU devices.

Even with the numerous publications on the topic of real-time FE on GPGPU devices, there exists an apparent lack of in-breadth testing and ex-

amination involving more, different, physical devices and solution strategies, a track followed here.

In the remainder of the section we present a short exposition on the background of soft tissue deformation methods, GPGPU technology evolution and the employed Compute Unified Device Architecture (CUDA).

1.2. Modeling for soft tissue applications

There currently exist a multitude of research papers dealing with deformable models used for surgical simulation, varying in levels of accuracy and speed of execution. The general rule of thumb is that more accurate methods require more computation time, as expected. One can say that at one end of the spectrum there are FEM-based models - very accurate yet prohibitively slow; on the other end of the spectrum the spring-mass model is fast and often-used for surgical training. However, fast heuristic models like the spring mass method (ref. [1], [2]) generally lack the ability to use nonlinear material models or material models in general beyond simple springs and dampers. For an overview of methods seen in literature, the reader is referred to [3], [4].

In line with the aforementioned requirements imposed by the biological domain under consideration one is compelled to use a solution method that is nonlinear both in terms of geometry and material behavior. The method of choice satisfying the requirements is a nonlinear finite element formulation capable of processing large deformations.

FE analysis of soft tissues for surgical simulation is an idea already pursued by several groups, most noticeably and in chronological order [5, 6, 7, 8, 9, 10]. These works also represent a sample of the evolution of research on real-time usage of the finite element method. Several approaches have been explored, both for implicit and explicit integration methods, static and variable topology of the mesh, different element types, orders and solution strategies. Publications on implicit FE are numerous, but perhaps the most notable early work can be found at [11] and a more recent [12].To the best of the authors' knowledge, Miller et al. (ref. [13], [14]) were the first to put forward work on an efficient nonlinear real-time FE solution to be used intraoperatively to provide information about tissue response on patient-specific anatomy during actual surgery. Their aim was to compute the final deformation field and track the position of critical elements within the mesh, e.g. a tumor. The solver was subsequently used by several groups, most recently [15].

1.3. A niche for GPUs

Regardless of the details of the method chosen, it is still evident that real-time FE analysis is computationally very demanding, and this is further exacerbated by the nonlinearity of the problem. Consequently, implementations of FE algorithms are subject to rigorous optimization strategies, most important of which being parallelization. Such a solution, using high performance computing devices and principles is widely adopted and most (if not all) commercial FE software packages support this feature, solving on multiple computing cores within a CPU, multiple CPUs within a single platform, or multiple platforms using a fast interconnect network. This type of parallelization can be considered coarse-grained, since generally a larger amount of work is done by individual units of the computing system.

Real-time computation on parallel or distributed systems demands favorable computation to communication time ratios. It also necessitates execution of problems of sufficient sizes to offset the latencies emerging from data transfers. The problem at hand, for which total solution times are essential, is of such size and nature that execution on a loosely-coupled system (e.g. CPU clusters) would exhibit high transfer latencies and stall computation. Finite element analysis is a non-arbitrarily divisible problem that imposes high communication requirements. Moreover, for our case of explicit analysis of limited size models, a high frequency of relatively small transactions is necessary. Such a problem benefits from having high memory throughput (in excess of 200GB/s on current GPU generations) and low latencies of a very tightly-coupled system such as a GPGPU.

1.4. Programming legacy GPUs

It is well known that there exists a significant difference between GPU and CPU hardware architecture. For specific problems this fact potentially leads to large increases in computational effectiveness [17]. Even older generations of graphics cards have higher computational throughput and memory bandwidth internally than current CPUs [16]. Utilization of GPUs to perform general purpose computation, especially linear algebra, started a decade ago by Göddeke [18] in 2004. GPUs, being inherently parallel machines, now in a considerably enhanced form, constitute a platform for parallelization of their own. Graphics processing units with a multitude of computing cores on a single board and on-board memory have very high arithmetic and memory throughput. Certain limitations do exist: memory capacity, accuracy and others elucidated further in the text. They are considered for

fine-grained parallelization and these principles and hardware are employed to speed up nonlinear FE computation. Their development and dissemination being driven by the gaming industry and making the hardware more accessible through frameworks such as OpenCL [19] and CUDA contributed decidedly to their increased use in scientific computing today. Harnessing the power of the GPU, however, was not straightforward. Prior to 2007 programming for the GPU was constrained to a more complex and indirect approach of inserting non-graphics code into the graphics pipeline of underlying purely graphical frameworks like OpenGL or Direct3D. Pixel shaders are an integral part of these graphics pipelines and are intended to execute in a highly parallel fashion, computing each pixel's RGBA color value. It is by writing shader programs that parallelization was exposed. It was, in effect, tricking the GPU to perform general computation instead of computing color values. For an excellent overview of pre-CUDA GPGPU computation see [20].

1.5. Compute Unified Device Architecture(CUDA)

The CUDA programming model uses kernels, essentially C language functions, that execute the same code on different data, analogously to a vector processor. Threads are organized in 1-,2- or 3-dimensional blocks, themselves generally consisting of one or multiple thread warps (a group of 32 synchronously executing threads). The total number of possible warps executing concurrently depends primarily on the memory requirements per-thread and is governed by a complex memory model involving architecture specific limitations on maximum runnable blocks, cfr. table 1. The CUDA memory model consists of several types of memory with different physical location (on and off the chip) and therefore different sizes, latencies and usage rules. Cards generally feature a two-level cache memory system. This proves very useful with data of high access frequency, or data with random memory locality or data dependent memory locality. Generally speaking, a problem that exhibits internal parallelism is decomposed into small work units that have no sequential dependency and can therefore be processed concurrently. It is up to the programmer to determine the optimal granularity of this decomposition, taking into account memory types, their capacity, rules of use and the algorithm at hand.

Other considerations while using CUDA involve: synchronization, memory layouts, throughput, latency and memory bottlenecks, etc. Large speedups can be achieved, but at this point in time, some understanding of the underlying architecture is required. This holds true especially of the interplay

GPU	K20c	C2075	GTX980	GTX780	GTX680	GTX580	GTX460
architecture(chip)	GK110	GF110	GM204	GK110	GK104	GF110	GF104
generation	Kepler	Fermi	Maxwell	Kepler	Kepler	Fermi	Fermi
cores/SM	192	32	128	192	192	32	48
registers/SM	64K	32K	64K	64K	64K	32K	32K
max blocks/SM	16	8	32	16	16	8	8
num of SMs	13	14	16	12	8	16	7
core clock[MHz]	706	1150	1126	863	1006	1554	1350
fp64/fp32 performance	1/3	1/2	1/32	1/24	1/24	1/8	1/12
memory bandwidth[GB/s]	208	144	224	288.4	192.3	192.4	115.2
L2 cache[KB]	1280	768	2048	1536	512	768	512

Table 1: Relevant information on the architectures of cards used.

between memory and instruction issue subsystems and the executing algorithm, since CUDA delegates much of the complexity back to the developer. The task is further hindered by the existence of different architecture generations that feature different amounts and even types of memory, different instruction issue mechanics etc. Herewith we avoid further functional detail on the CUDA architecture as the work focuses more on testing than implementation details. Some technical detail is necessary, however, and will be mentioned alongside associated observations further in the text. Necessary information about the architecture of GPUs used is available in table 1. For the interested reader a detailed overview of CUDA is available through [16], and [21] for advanced material.

This paper presents an implementation of the Total Lagrangian Explicit Dynamic algorithm on modern GPGPUs using CUDA. We use a range of GPUs including the current (as of the time of the writing) Maxwell and previous Kepler and Fermi architectures. A single/double precision floating point computation study is performed to examine the accumulation and evolution of the round-off error and its impact on accuracy and execution time. Similarly, the tests have been performed on a range of model sizes to examine the scaling of performance. Furthermore, each simulation is run with and without the texture memory space and the presence or absence of additional material complexity (tissue damage model) and its impact is commented upon. An industry-proven finite element code Abaqus is used as the

ground truth in terms of accuracy, as well as a baseline to measure speedup against. In section 2 details are presented of the algorithm and implementation, hardware, example problems and solution regimes, followed by results (3), discussion (4) and conclusions (5) on the topic.

2. Materials and methods

2.1. Total Lagrangian Explicit Dynamic

After finite element spatial discretization using the total Lagrangian (TL) formulation, we arrive at the familiar $\mathbf{P}(\mathbf{u}) = \mathbf{R}$ system of equations, where $\mathbf{P}$, $\mathbf{u}$,and $\mathbf{R}$ are the internal nodal forces, displacements and external forces, respectively. The central difference scheme is used for temporal discretization and drives the simulation forward in the time domain. Since the aim is to solve a static problem using a dynamic (explicit) method, mass and damping contributions are added into the equation:

$$\mathbf{M}\ddot{\mathbf{u}} + q\mathbf{M}\dot{\mathbf{u}} + \mathbf{P}(\mathbf{u}) = \mathbf{R}, \tag{1}$$

where $\mathbf{M}$ is the diagonalized mass matrix. In the notation throughout, the left superscript denotes time-step and the right subscript denotes element scope. By using the familiar constant step central difference formulas, we define the following temporal derivatives:

$${}^{t}\dot{\mathbf{u}} = \frac{{}^{t+\Delta t}\mathbf{u} - {}^{t-\Delta t}\mathbf{u}}{2\Delta t}; {}^{t}\ddot{\mathbf{u}} = \frac{{}^{t+\Delta t}\mathbf{u} - 2{}^{t}\mathbf{u} + {}^{t-\Delta t}\mathbf{u}}{2\Delta t^2} \tag{2}$$

By combining 1 and 2 and with some algebraic manipulations around the known forces and the sought displacements:

$${}^{t+\Delta t}\mathbf{u} = a(\mathbf{R} - \mathbf{P}) + b{}^{t}\mathbf{u} - c{}^{t-\Delta t}\mathbf{u}; \tag{3}$$

Where a, b, c are the final central difference coefficients, C_r is the convergence rate of the central difference scheme set close to unity and q is the damping coefficient and its expression we opted for:

$$a = \frac{2\Delta t^2}{(2 + q\Delta t)M_e} \qquad b = 1 + \frac{(2 - q\Delta t)}{2 + q\Delta t} \tag{4}$$

$$c = \frac{2 - q\Delta t}{2 + q\Delta t} \qquad q = \frac{2(1 - {C_r}^2)}{\Delta t(1 + {C_r}^2)} \tag{5}$$

Further detail can be found in: [13, 22]. A pseudo-code of the solution of the system in equation 1 is provided below.

Pre-computation phase:

1. Compute shape function derivatives in initial global coordinates (the reference configuration in total Lagrangian) from natural isoparametric coordinates $\nabla \mathbf{N}_{Iso}$ and initial nodal positions $\mathbf{p}_e$, using the Jacobian matrix ${}^0\mathbf{J}_e$:

$$ {}^0\mathbf{J}_e = \frac{\partial(x, y, z)}{\partial(\xi, \eta, \zeta)} = \nabla \mathbf{N}_{Iso} \mathbf{p}_e \tag{6} $$

$$ \nabla \mathbf{N}_{Glo,e} = {}^0\mathbf{J}_e^{-1} \nabla \mathbf{N}_{Iso} \tag{7} $$

2. Assemble initial strain-displacement matrix: ${}^0\mathbf{B}_e$
3. Compute element volumes and diagonalized mass matrix $\mathbf{M} = M_e \mathbf{I}$
4. Compute the central difference coefficients a, b, c and damping factor q from equations 4.

Iteration phase. For every time point t:

5. Compute the deformation gradient using the shape function derivatives and current displacements ${}^t\mathbf{u}_e$:

$$ {}^t\mathbf{F}_e = \mathbf{I} + \nabla \mathbf{N}_{Glo,e} {}^t\mathbf{u}_e \tag{8} $$

6. Compute the right Cauchy-Green deformation tensor, Jacobian determinant:

$$ {}^t\mathbf{C}_e = {}^t\mathbf{F}_e^T {}^t\mathbf{F}_e \ , \ {}^tJ_e = det({}^t\mathbf{F}_e) \tag{9} $$

7. Compute the Second Piola-Kirchhoff stress:

$$ {}^t\mathbf{S}_e = 2 \frac{\partial^t \mathbf{\Psi}_e}{\partial^t \mathbf{C}_e} \tag{10} $$

8. Compute force contributions. Total Lagrangian expression for force computation, using 1 point Gaussian (under-)integration:

$$ {}^t\mathbf{P}_e = \int_{V_0} {}^0\mathbf{B}_e {}^t\mathbf{F}_e^T {}^t\mathbf{S}_e \mathrm{d}V_0 = 8 det({}^0J_e) {}^0\mathbf{B}_e {}^t\mathbf{F}^T {}^t\mathbf{S}_e \tag{11} $$

9. Summation of elemental contributions into the global force vector:

$$ {}^t\mathbf{P} = \sum {}^t\mathbf{P}_e \tag{12} $$

10. Obtain new displacements ${}^{t+\Delta t}\mathbf{u}$ using eq. 3

11. Impose boundary conditions for the next step:

$$ {}^{t+\Delta t}\mathbf{u}_{BC} = {}^{t+\Delta t}\mathbf{u}_{BC}^{imposed} \tag{13} $$

$$ {}^{t+\Delta t}\mathbf{R}_{BC} = {}^{t+\Delta t}\mathbf{R}_{BC}^{imposed} \tag{14} $$

12. Advance to the next step: $t \leftarrow t + \Delta t$

In light of increasing the speed of the algorithm, pre-computing all possible values is essential. Due to the total Lagrangian spatial discretization scheme chosen, all derivatives with respect to the original (reference) configuration do not change. The shape function derivatives, therefore, do not change across time-steps and are computed in the first stage. The central difference method coefficients are also pre-computed as they too do not change throughout.

Note that since the mass matrix is diagonal, the equation in step 10 is algebraic and each solution displacement vector may be computed independently. This is, in fact the originating point of much of the increased speed provided and is fundamental to explicit formulations. An optimal convergence rate, especially for dynamic relaxation has been discussed in [23], indicating an additional potential optimization in terms of reduced number of executed steps, an idea not presently pursued.

The contributions to the global force vector are computed in step 8. The force contributions are summed in the following step. Single point Gaussian integration is employed within the 8-node under-integrated hexahedra to obtain a simple expression for nodal forces from element stress in step 8. Note that here we implicitly update the strain-displacement matrix with the current deformation gradient, at every time-step. The element stress was in turn computed from the right Cauchy-Green tensor and deformation gradient in steps 5 to 7. Depending on the material model used, the stress computation in step 7 can vary. In our implementation, Neo-Hookean material with and without damage are used, as explained in section 2.2. The nodal displacements are calculated using the resultants and the central difference scheme. Finally, new displacement boundary conditions are assigned to the boundary nodes, to prepare for the next step. For a more detailed discussion on TLED, the reader is referred to [13] and subsequent work by the group.

The critical time-step was computed by using linear theory [24] and honors the Courant-Friedrichs-Lewy (CFL) condition. A costly modal analysis of the domain is avoided this way, and it is assumed that the following equations with an appropriate weighting factor are sufficient. A weighting factor (Courant number) of 0.8 was used in the present case but generally depends highly on the problem and requires some engineering insight. E.g. presence of contact or highly nonlinear behavior of material or loading curve would necessitate a lower weighting factor.

$$\Delta t = \frac{L_e}{c}; L_e = \frac{V_e}{A_e}; c = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{15}$$

Where L_e is the characteristic length, computed using the smallest element area in the model A_e and its volume V_e, ρ is the mass density of the element.

2.2. Material model

The material model employed is a 3 dimensional hyperelastic Neo-Hookean model, often used for soft tissues. The Neo-Hookean is conceptually relatively simple and easy to implement, especially using the proposed computational framework. A strain energy density function (SEDF) can be defined by an additive split into a deviatoric and a volumetric term:

$$\Psi = \frac{\mu}{2}(\bar{\mathbf{I}}_1 - 3) + \frac{\kappa}{2}(J-1)^2, \tag{16}$$

where μ and κ are material constants - shear and bulk modulus, respectively, cfr. table 2. $\bar{\mathbf{I}}_1 = J^{-2/3}\mathbf{I}_1$ is the first invariant of the deviatoric part of the right Cauchy-Green deformation tensor $\mathbf{C}$. The second Piola-Kirchhoff stress tensor $\mathbf{S}$ can be derived from the SEDF according to step 7 in section 2.1.

By using the chain and product rules, and the known:

$$\frac{\partial \mathbf{I}_1}{\partial \mathbf{C}} = \mathbf{I} \text{ and } \frac{\partial J}{\partial \mathbf{C}} = \frac{1}{2} J \mathbf{C}^{-1} \tag{17}$$

the final form of the Second Piola-Kirchhoff stress is obtained:

$$\mathbf{S} = 2\frac{\partial \Psi}{\partial \mathbf{C}} = \mu J^{-\frac{2}{3}}(\mathbf{I} - \frac{tr(\mathbf{C})}{3}\mathbf{C}^{-1}) + \kappa J(J-1)\mathbf{C}^{-1} \tag{18}$$

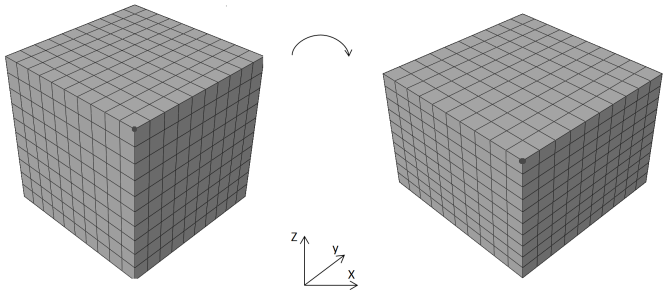


Figure 1: Initial and deformed geometry of a 10x10x10 model

$$\Psi^{dev} = (1-d)\hat{\Psi}^{dev} \tag{19}$$

$$d = \gamma[1 - exp(-\beta/m)] \tag{20}$$

Where $\hat{\Psi}$ is the undamaged deviatoric strain energy, $\gamma \in]0, 1]$ is a weighting factor and m is a parameter of the damage model. The actual damage variable is d, indicating the loss of integrity of the material on a scale from 0 to 1. β is the maximum value of energy in the interval $0 < t < \tau$ initially set to zero:

$$\beta = \underset{0<t<\tau}{max}(\hat{\Psi}(t) - \Psi_0) \tag{21}$$

The value for Ψ_0 is set to an experimentally obtained value above which damage is initiated. In the case of arterial tissue, for example, the strain energy at systolic pressure can be used. Using this adapted strain-energy function, the second Piola-Kirchhoff stress in the damaged material $\mathbf{S}^{dam}$ becomes simply:

$$\mathbf{S}^{dam} = (1-d)\mathbf{S} \tag{22}$$

2.3. CUDA implementation

The pseudocode described in section 2.1 is implemented using Nvidia CUDA. An overarching and somewhat obvious factor in speeding up code is replacing loops with parallel functions (kernels). However, this can only be done if iterations are independent from one another. For this reason parallelization is done over elements and nodes rather than time-steps.

The solver has been ported to the GPU entirely. The host CPU serves only to iterate through time-steps invoking the respective kernels, swapping pointers and performing other trivial work. The GPU handles the entire volume of calculation described in steps 5 to 11. The parallelization of the algorithm is achieved by splitting the work into three functions: the computation of elemental forces, assembly of the global force matrix and computation of new displacements, and finally the imposition of new displacement boundary conditions. Costly communication between the CPU and the GPU depends on the higher latency and lower bandwidth of the PCI bus, and is kept to an absolute minimum, transferring data only at the very start and end of a simulation.

The first kernel runs one element per thread and computes equations in step 5 to 8. It receives input data on: element connectivity, shape function derivatives, material model parameters, damage parameters, volumes, etc. The output of the routine are individual force contribution vectors of each node, for all elements. The damage computations are included in this function since the strain energy and stress are computed here. Computing force contributions is the most demanding part of the algorithm, taking about 70% of total solution time. With high internal memory requirements to store input data as well as the derived values (like the deformation gradient or stresses) the parallelization level is significantly reduced. Consequently the hardware will allow and maintain only a relatively small amount of threads in flight, e.g. 3584 elements are processed concurrently on the C2075 device in the code. Note that individual force contribution vectors from adjacent elements of each node are stored separately instead of summed immediately to avoid storage serialization or simply incorrect results due to the non-atomic nature of CUDA global memory writes as dedicated global atomic operations have not been used. If required, damage computation is performed in this kernel, between steps 7 and 8. In the case that damage-corrected behavior is requested, for relatively few memory fetches (Ψ_0 and β) per element we can examine the computation time response. Note that the computation of the strain energy (eq. 18) is only performed in this case.

The second kernel operates on a per-node basis and processes steps 9 and 10. Initially, the computed force contributions from the surrounding elements are accumulated and summed. The aggregate forces are used immediately in the central difference scheme without explicit storage. Here, the precomputed central difference coefficients a, b and c are used. The resulting displacement vectors are again per-node values and are stored to global memory for use in

Parameter	Value
μ	1006.7e-6 MPa
κ	50e-3 MPa
λ	49.3e-3 MPa
γ	0.9 [-]
m	0.004 [-]
C_r	0.995 [-]
Ψ_0	0 [-]

Table 2: Material, damping and damage parameters.

the following kernel. The imposition of force loading is applied in step 9 in the developed code, after summation.

The last and in many ways the simplest kernel (step 11) performs the trivial task of imposing displacement or force boundary conditions by artificially modifying associated vectors of relevant nodes. Note that another approach is possible and was explored comprehensively in [25] where kernel 2 and 3 were merged into one due to the same granularity of the kernels. Due to the much lower number of state variables associated with nodes in the two node-granular kernels, they are generally never memory-bound. Since they do not take up much of the execution time they are much less interesting from an optimization point of view.

A special type of cached global memory access called texture memory or texture fetching is provided to the programmer in part to facilitate the handling of data with irregular or random memory locality. In our algorithm, the nodal displacement data depend on element connectivity information and therefore has irregular spatial locality in memory. Consequently, our code includes toggling of texture memory functionality, applied only on nodal displacement data. Similarly, the code includes the toggling of damage subroutines for easy measurement of the computational impact of including damage calculation. These devices also have both single and double precision computing cores, which are physically distinct. The number of double precision cores is generally a fraction of their single precision counterparts, exact numbers depending on the device generation. Due to this fact, computation using double precision is considerably slower (cfr. table 1) at peak saturation but enables more accurate results, as examined and discussed in section 4.

no. of elements	125	1000	3375	8000	15652	42875	91125
no. of time-steps	6706	13435	20153	26870	33588	46875	60484
ABQ/Explicit[hms]	1s	12s	1m10s	3m47s	9m08s	41m22s	1h39m37s
K20c(tex)[s]	0.417 / 0.493	0.822 / 0.983	1.761 / 2.290	4.253 / 5.991	8.579 / 13.010	28.324 / 42.546	73.599 / 111.427
K20c(no tex)[s]	0.418 / 0.495	0.812 / 0.987	1.769 / 2.405	4.295 / 6.199	8.718 / 13.588	29.063 / 44.552	75.824 / 117.050
C2075(tex)[s]	0.326 / 0.342	0.658 / 0.923	1.670 / 3.700	5.154 / 11.995	10.759 / 25.120	35.326 / 85.660	93.364 / 216.315
C2075(notex)[s]	0.328 / 0.353	0.683 / 0.947	1.775 / 3.657	5.748 / 10.642	11.958 / 24.144	38.944 / 80.616	101.675 / 204.821
GTX980(tex)[s]	0.149 / 0.432	0.328 / 1.129	0.817 / 2.854	1.960 / 10.488	4.384 / 20.534	14.579 / 68.252	34.143 / 181.171
GTX980(notex)[s]	0.135 / 0.452	0.313 / 1.133	0.777 / 2.798	2.030 / 7.637	4.962 / 18.684	15.899 / 66.755	39.950 / 178.829
GTX780(tex)[s]	0.255 / 0.729	0.519 / 1.463	1.163 / 4.597	3.019 / 11.972	6.113 / 26.290	20.275 / 91.132	53.097 / 243.660
GTX780(notex)[s]	0.253 / 0.729	0.508 / 1.461	1.161 / 4.591	3.036 / 12.020	6.197 / 26.606	20.681 / 92.697	34.330 / 248.258
GTX680(tex)[s]	0.328 / 0.531	0.682 / 1.200	1.554 / 3.710	4.398 / 11.515	10.048 / 26.673	35.479 / 89.296	92.710 / 231.590
GTX680(notex)[s]	0.340 / 0.511	0.715 / 1.186	1.579 / 2.822	4.492 / 11.089	10.301 / 26.055	36.873 / 89.754	92.659 / 235.876
GTX580(tex)[s]	0.241 / 0.255	0.487 / 0.634	1.138 / 2.295	3.210 / 6.997	7.205 / 15.918	24.323 / 49.690	60.464 / 121.962
GTX580(notex)[s]	0.247 / 0.275	0.509 / 0.673	1.202 / 2.311	3.384 / 6.958	7.807 / 15.372	26.617 / 48.550	66.074 / 119.844
GTX460(tex)[s]	0.330 / 0.354	0.707 / 1.302	2.570 / 5.323	7.647 / 15.376	16.938 / 33.994	60.054 / 117.197	158.777 / 305.457
GTX460(notex)[s]	0.309 / 0.364	0.738 / 1.324	2.746 / 5.366	8.164 / 15.047	17.956 / 33.116	52.321 / 115.505	163.402 / 306.171

Table 3: Run times of the base Abaqus solutions and our implementation with and without textures, using single/double precision. These are CUDA per time-step solution times multiplied with Abaqus' number of steps.

2.4. Example problem

The testing regimes employed in this study are all performed on the uniform compression of a simple cube model (fig. 1). Cube meshes of 50x50x50mm with varying mesh density have been created for this purpose. The above mentioned Neo-Hookean material model was applied to all the elements of the mesh and stable time-steps have been computed according to equations in section 2.1.

The simple boundary conditions applied prescribe no displacements in the axial direction (Z) for the bottom nodes and enforces displacements along the same axis for the top nodes, all other degrees of freedom are unconstrained. The load was applied gradually along a smooth loading curve up to 20% strain. This enabled us to minimize inertial effects and to study the phenomena under quasi-static conditions. The 5 second loading time was distributed along around 6700 to more than 60400 steps in different models.

The solution (fig. 1 on the right) is homogenous in the stress distribution. For this simple problem, an even coarser mesh or an analytical solution would essentially provide the same resulting solution field - the accuracy does not increase with mesh refinement. This intentional decision is made for the purpose of easier mesh generation and error comparisons as well as the lack of complicating effects such as potential loss of stability due to subdivisions of complex meshes or a likely additional reduction of step size in case of a few poorly formed elements. These issues would somewhat occlude the effect of parallelization of TLED, our primary goal.

Simulations were run multiple times with a different combination of settings. These solution regimes are designed to provide a multitude of results and test the algorithm and the hardware under different conditions.

The round-off errors were evaluated by the difference between GPU solutions only.

2.5. Solution regimes

Extensive testing of the described algorithm has been conducted encompassing a variety of changing conditions. Since different GPUs exhibit different internal hardware and methods, the algorithm was executed and data collected on the Nvidia devices listed in table 1. The list contains devices from the latest Maxwell [26], and preceding Kepler [27] and Fermi generation [28]. As opposed to older generation cards, all of the mentioned GPUs have double precision floating-point computation capability.

The simulation code is extensively templated to toggle among different solution regimes, including those between single (fp32) and double precision (fp64) for comparison in terms of speed, accuracy and the evolution of the round-off error. The templated device and host code data structures and kernels also assure that the compiler optimizes away all unnecessary branch checks when running a particular combination of parameters in a solution regime, i.e. double precision with textures and damage code in a single kernel. An important reduction in thread divergence and consequent slowdown is avoided this way and presently there is no divergence anywhere in the code.

The existence or absence of additional material complexity (damage in our case) is also included in the testing regimes to examine the effect of the additional arithmetic and memory requirements in the kernels, specifically for the described simple damage model.

Similarly, the usage of texture memory fetching on displacement variables is also set as a templated boolean value so performance can be tested with or without this special memory space.

In addition, to understand the performance scaling of the algorithm, tests were performed on meshes of varying density, ranging from 5 elements to 45 elements per edge and totaling from 125 to 91125 elements.

As mentioned, respective Abaqus solutions were used to verify each of the solutions performed by the described custom solver. The execution parameters used for Abaqus simulations are single core, double precision with all other additional settings at default. Furthermore, in contrast to our solution, Abaqus does use a smaller time-step with a fixed ratio for all simulations of 2/3. All comparative results between the two solvers are corrected for this fact.

Control of floating point precision, material complexity in terms of damage, and texture memory space usage are mutually inclusive and constitute eight employed execution combinations. These have been tested on all GPUs and models. All results except damage inclusion have been compared to respective Abaqus solutions. Identical boundary conditions and smooth loading curve were used by both solvers as well as the material parameters as explained in section 2.2. Abaqus jobs were run on a system with an Intel Xeon E5-2630 processor and 128GB of RAM, using only one core, double precision and a built-in identical material model. Results for the GTX980 and GTX780 were obtained using CUDA nvcc v6.5 compiler and Visual Studio 2013 C++ compiler, and nvcc v5.5 and Visual Studio 2008 C++ for older

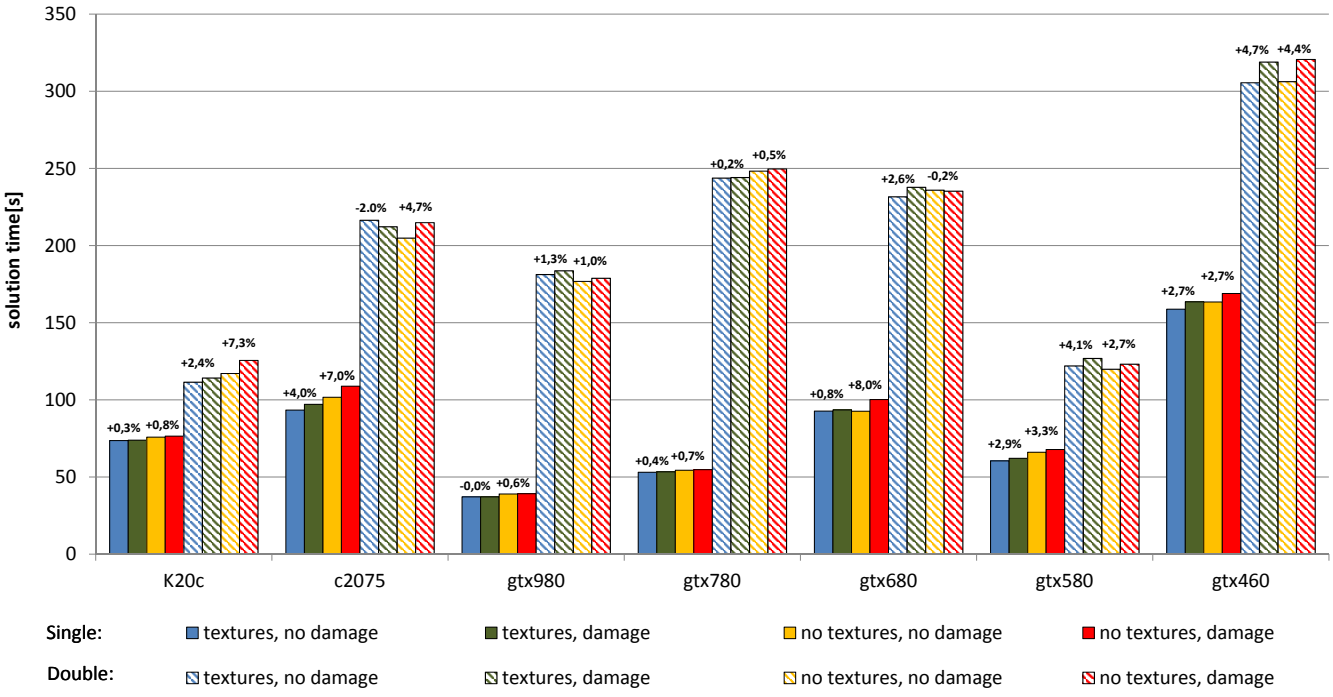


Figure 2: Total solution times for the largest model, involving 91125 elements over 60400 time-steps on all GPUs. The effect of damage is included above the related column pairs. These CUDA results are corrected for the time-step discrepancy between the two solvers.

cards. The duration of the Abaqus solutions were taken from its output files, while timing CUDA code was performed on a high-accuracy hardware clock accessed through the dedicated CUDA built-in functions. The precomputation phase of the algorithm is not considered significant in its duration with respect to the iteration phase and is not timed.

3. Results

The time-step run-time of Abaqus simulations were obtained by dividing the total CPU time with the known number of steps, analogously to the CUDA simulations which are, however, subsequently corrected for the mentioned relation $\Delta t_{abq} = 2/3\Delta t_{cuda}$ in the relevant results. The accuracy of the newly implemented algorithm was evaluated by comparing the displacement vector in the XY plane of one top node at the corner. For all simulations the deviation from Abaqus solutions never exceeds 0.3% on any node, in terms of

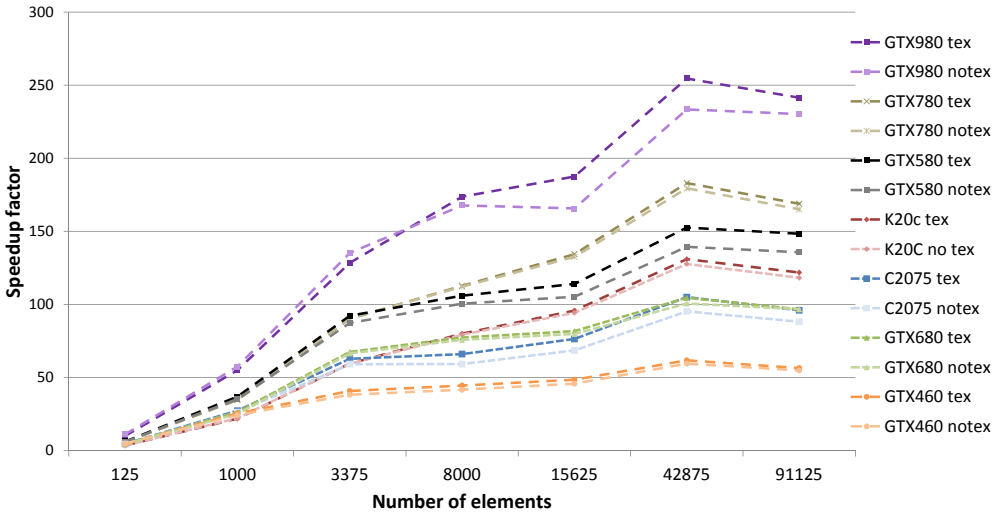


Figure 3: Speed up factors versus Abaqus as a function of model element density. These factors compare durations of a single time-step in single precision.

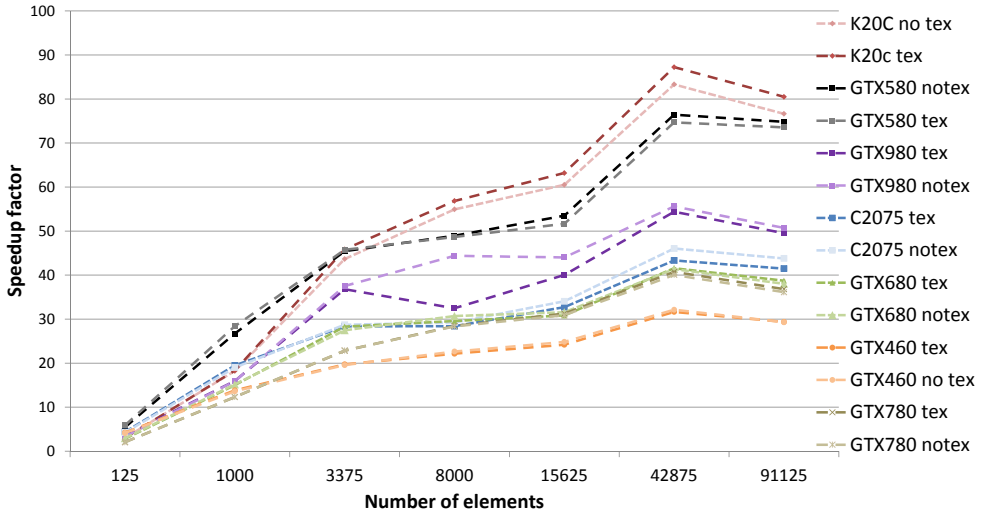


Figure 4: Speed up factors versus Abaqus as a function of model element density. These factors compare durations of a single time-step in double precision.

GPU	no tex no dam	tex no dam	no tex dam	tex dam	average	perform. at peak
$K20c$	64.8%	66.1%	60.9%	64.7%	64.1%	33.3%
$C2075$	49.6%	43.1%	50.7%	45.8%	47.3%	50.0%
$GTX980$	22.0%	20.5%	22.0%	20.2%	21.2%	3.1%
$GTX780$	21.9%	21.8%	21.9%	21.8%	21.9%	4.2%
$GTX680$	39.3%	40.0%	42.6%	39.3%	40.3%	4.2%
$GTX580$	55.1%	49.6%	55.1%	48.9%	52.2%	12.5%
$GTX460$	53.3%	52.0%	52.7%	51.3%	52.3%	8.3%

Table 4: Ratio of solution speeds of double precision runs with and without textures and damage to their single precision counterpart, in percentage. The averages give general information on how well the hardware handles double precision against single for this specific algorithm. The last column shows the theoretical performance for a fully saturated device at peak arithmetic throughput.

the length of displacement vectors, throughout the simulations. For detailed error analysis in tension, compression and shear using TLED we refer the reader to [25]. The initial configuration and the loaded, converged configuration are symmetric with respect to all orthogonal planes and vertical edges remain vertical.

Table 3 shows the solution times for the baseline Abaqus simulations compared to GPU solutions, both without damage subroutines. Figure 2 shows all combinations of execution parameters on all GPUs for the densest mesh. Since Abaqus and TLED solutions use different time-step sizes, different number of steps are necessary to reach the end of the simulation ($t = 5s$) for the two machines. In order to provide an approximate measure of speedup of total simulation times and per time-step run-times, CUDA results in table 2, table 3 and figure 3 are corrected for this discrepancy by increasing the CUDA runtime using the mentioned relation.

In terms of performance, the GPUs follow roughly a natural order: the latest GPUs are faster than older ones, with the possible exception of the GTX580. It achieves high speedups overall, mostly due to its comparatively very high clock rate and its excellent fp64/fp32 peak ratio, given it is primarily a gaming card. Note that some tests were done on computers with different CPU configurations for practical purposes. The difference in performance due to this is not considered significant in the research since the totality of compute-intensive code is performed on the noted GPUs, on stock

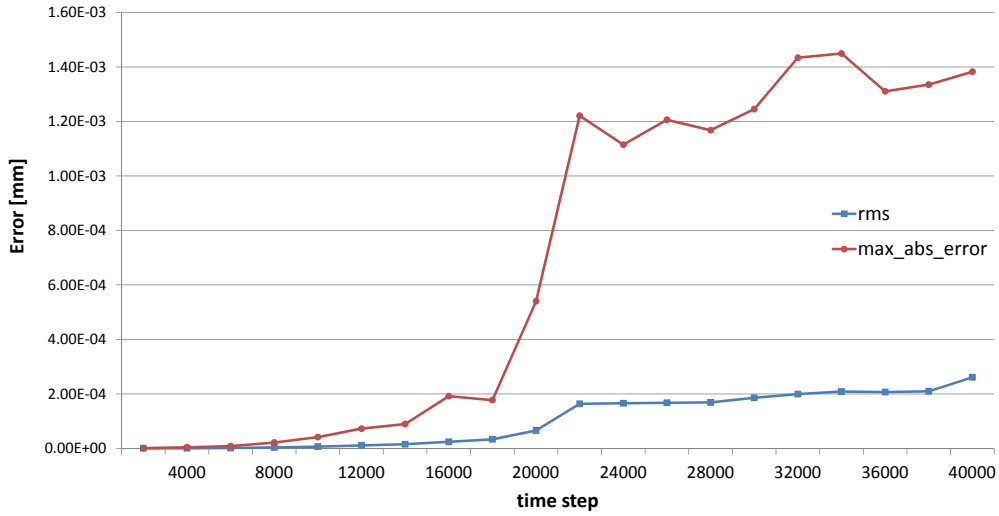


Figure 5: Evolution of the root mean square error and the maximum absolute error calculated on the set of Euclidean distances between the final solution steps of single and double precision simulations. This analysis was performed on the largest mesh containing 97336 nodes. The percentage errors are in relation to the final displacement vectors of the solution.

clock speeds.

Expectedly, the precision of number representation has the largest impact on solution times for the same mesh, as can be seen in table 3, figure 2 and in the difference between figures 3 and 4. Table 4 shows that the K20c device has the highest effective single to double performance ratio, with average double precision computational throughput of about 65% of its single precision counterpart. In contrast, the least favorable GPU performance in terms of double precision in relation to its single precision are the GTX980 and the GTX780. Further detail is provided in the following section. The impact in terms of accuracy of using double precision number representation is illustrated in figure 5. The computation of accuracy was performed on the set of all resulting double precision vectors of the largest model with respect to their single precision counterparts throughout the length of the simulation. To understand this performance/accuracy tradeoff, the largest model is analyzed since it involves the largest number of time-steps to converge and therefore more opportunity to accumulate the round-off error. Aside from the number of time-steps, the error also accumulates due to the larger number

of nodes through which force waves propagate. It is safe to say that other models in this study have round-off errors that are smaller. Additionally, since displacements are at the end of the computational chain per time-step it is also safe to say that the round-off error associated with the stress field or other variables is never higher.

The memory requirements for damage processing is rather small: only two global memory fetches (β_{max} and Ψ_0) and stores. Relatively few compute operations are needed as well, essentially a computation of current strain energy $\hat{\Psi}^{dev}$ and a comparison to the initial energy. Since the loading and computation only involve scalars (no matrix multiplication or inversion), the performance impact is expected to be small. The measured impact however is hard to justify and explain from this perspective, and varies - as seen in figure 2. The effect is shown above the column pairs that use or do not use damage. Across all solutions, the impact varies from almost no impact (0.3% single precision on K20c) to a relatively large impact (8% single precision on the GTX680) and even a positive impact (2% speedup on double precision c2075) as shown above the respective column pairs in table 2. This is a consequence of the lack of architecture-specific optimizations to exploit the different mechanisms of caching and the amounts of cache space (especially L2), compute capabilities of the cards used and the sometimes unpredictable compiler behavior. For a more uniform response a larger amount of computation and memory operations would be necessary and therefore precise reasons for this apparently erratic behavior would necessitate more detailed profiling of individual devices and a much more gradual increase in mesh density.

In the test cases, the effect of using texture memory was not significant as can be seen in table 3, and figure 2. Since CUDA does not support native double precision floating point texture fetching, the code uses two 16 byte int4 values per each double4 value. Using this available makeshift functionality impacts performance on almost all cards and suggests against double precision texture fetching, clearly visible in figure 4 as well. A slight improvement in performance can be observed by using texture memory in Fermi and Maxwell GPUs in single precision (especially at denser meshes), and a disadvantage or similar results in double precision computation.

As is most clear from table 3, for the smaller meshes in the testing cases the speed-up difference between the GPUs is marginal. This is due to the fact that the devices instruction and memory pipelines are not saturated at that point and the device is underused. The saturation point depends on

the implementation and the material model but is mainly due to the mesh density and architectural differences: number of cores, amount of memory available, memory thresholds, etc. Therefore, the results indicate that for smaller meshes, it makes little difference which GPU is used for the simulation. The higher density models discriminate more clearly between the devices and a familiar strong scaling curve of parallelization can be observed in figures 3 and 4.

The results in figure 3 show a speed-up factor of TLED versus Abaqus on a per-time step basis using textures. The total solution time speed-ups are proportional to these values and range from 1 second to 1h39m37s for Abaqus and from 0.1s to 39s for the parallel solver on the fastest GTX980 device, in single precision.

4. Discussion & future work

Section 3 provides measured solution times for simulations of varying element density, solution regimes and GPUs with encouraging results. Note that solution times were compared to those obtained with a commercial FE software Abaqus, for a simple, analytically solvable example problem. This allowed us to make a clean comparison between the solvers and the GPUs themselves, without having to include the technicalities of a more complex simulation (e.g. contacts, follower loads). Hence, with the available information, it is now possible to make a tentative selection of the preferred model size for an actual FE problem. For example: an FE mesh consisting of 8000 elements advanced through 25000 time-steps can be solved within approximately 2 seconds (table 3). However, certain limitations and caveats do apply and we note them in this section.

In this simple static model problem, an implicit solution would almost certainly perform faster both in the GPU and CPU setting. We opted for using and parallelizing an explicit solver as it can be used more naturally in a dynamic simulation environment, i.e. virtual reality surgery. Additionally, the accumulation of total damage or element-level damage accumulated in the tissue may be detected (by way of pre-defined thresholds) more readily, given smaller time-steps of the explicit method.

4.1. Solver and card architecture

This particular choice of element technology, the underintegrated 8-node hexahedron, is well-known for its fast computation due to a constant strain

field, low field variable (degrees of freedom) requirements, a single integration point and therefore a low total number of operations to produce nodal force vectors. These benefits are also supported by the particularities of implementation and solution on a CUDA GPGPU. Arguably the first more complex element, a fully integrated "serendipity" 20-node hexahedron outweighs the underintegrated hexahedron almost 3 times in terms of the displacement degrees of freedom and the shape function derivatives alone, affecting steps 5-7 in the algorithm. Significantly exacerbating the problem is the fact these computations now need to be performed 8 times, one per integration point, to compute step 8. However, aside from pure FE methodology, using the 20 node hexahedron will have adverse effects on the CUDA devices computational time, which will not increase proportionally. As memory requirements increase to store additional degrees of freedom, kinematic and other intermediate variables, register pressure (the fastest and most scarce memory type) is increased causing registers to spill out into global memory (slowest and largest memory space), potentially pollute the L1 and L2 caches and decrease the overall execution speed further. The phenomenon of register spilling occurs at different memory usage thresholds and is GPU-dependent. Nevertheless, it is well worth taking into consideration and is an important variable in the process of deciding which elements to use. As is in our case, this difficulty is present in Lagrangian descriptions due to the changing nodal position variables with respect to a reference configuration, which need to be stored and tracked. It is important to note that the Abaqus program was used foremost as the ground truth in terms of accuracy and as a baseline to compare solution times only for a very simplistic scenario that includes a simple loading curve and material, absence of contacts etc.

Maxwell and Kepler cards are substantially different to Fermi in terms of several architecture design decisions. The differences that relate to this study are primarily in the maximum number of thread blocks per streaming multiprocessor: 8 per Fermi SM and 16 for Kepler SM (or SMX), cfr. table 1. With 32 threads run per block for the force computation kernel this change increases the number of threads in flight and directly affects the number of elements or nodes processed concurrently, in a single pass. This number also depends on the number of available streaming multiprocessors, specific to the card, and ultimately yields: 1792 concurrent threads for the GTX460, 3584(C2075), 4096(GTX580) for Fermi, 4096(GTX680), 4992(K20c) and 4608(GTX780) for Kepler cards and 6144 threads for the Maxwell GTX980. The numbers of concurrently running threads generally

indicate better final performance (apparent in all results) as a smaller number of passes is necessary to process all elements or nodes of a time-step. However, the numbers should still be offset for other architectural decisions. The number of registers per thread available on Fermi cards is larger than on Kepler and Maxwell, despite the doubling (from 32K to 64K) of the register file, due to the an even larger increase in number of cores (32 to 192/128 per SM for Kepler/Maxwell). This change increases local memory usage (through register spilling) on Kepler and Maxwell and stalls execution due to frequent, implicit, fetches from L1, L2 and the most detrimental - global memory. Fortunately, GK110 and later chips have a much larger L2 pool (as well as memory bandwidth) and different management modes of on-chip L1/shared memory reducing spillage into global memory, consequently alleviating the effect.

Register spilling occurs to a slightly smaller degree on Maxwell versus Kepler due to its lower number of cores per SM. Note that Kepler GK110 chip does increase the maximum registers usable per thread from 63 on Fermi and GK104 to 255 on the GK110, identical to GM204. In the present case this change manifests itself by reducing the e.g. K20c number of active threads from 6656 to the reported value of 4992, since the threads' register requirements are replaced by the maximum allowed number of blocks per SM as the primary execution limiter. The reason for this is the large memory requirements of the forces computation kernel, the most impactful of all the kernels, requiring approximately 130 registers without and 135 with damage. If fp64 is used the number of registers needed are roughly doubled. There is no thread divergence, atomic operations or shared memory usage present anywhere in the code - these would impact performance more on the Fermi architectures.

Generally speaking, the occupancy (ratio of currently running and maximum runnable threads) of the devices is rather low, which (in a majority of cases) indicates that better utilization of the device is possible, but not achieved for the specific algorithm or implementation. The TLED algorithm has large memory requirements per thread, making memory an important limiter of occupancy and latency as the primary stall reason. If it would be possible to increase the overall granularity of the algorithm to something smaller than one element per thread, provided an accompanying reduction in memory requirements, it would be possible to obtain higher occupancy and better overall performance.

In terms of the relevant GPU architectural differences Kepler-,Maxwell-

specific features like Dynamic Parallelism (ref. [29]), HyperQ, or the 48kB read-only data cache have not been used in the code. Dynamic Parallelism technology removes the need to transfer execution control or data between host system and the CUDA GPU. Consequently there is potential to reduce or remove the kernel invocation ($\sim 10\mu s$) time on Kepler and future hardware. Given the high number of kernel invocations owing to explicit integration, removing invocation times and letting the GPU manage itself could prove beneficial in terms of total solution times.

4.2. Precision, texture memory and damage

Ideally, the difference in the performance of single and double precision computation is influenced primarily by ratios of fp32 to fp64 computing cores present in different chip architectures, shown in table 1 and are generally better for professional cards. These official ratios are not reflected in the measured results (table 4) due to the nature of the algorithm being executed and its interplay with the architecture. Performance ratios are larger than what they would be at peak saturation (also shown in 1) of arithmetic pipelines, indicating that single and double precision cores are indeed stalled and underutilized. Exacerbating the problem is the fact that double precision operands are twice the size and require more transactions. Counterintuitively, in the present case this is beneficial and does not incur such a high cost in final fp64 performance. In conclusion: the compute performance in our case is latency-bound by the memory subsystem (and intricacies thereof) and therefore table 4 represents realistic performance expectations for such a memory-transaction-heavy code ex post facto.

It is worth noting that power-consumption is most likely not an as important design priority for gaming cards as it is for professional ones (a point that also stands for single/double precision performance in favor of professional cards) but is considered not pertinent for the current study.

If the round-off error is considered acceptable, it is evident that great speedups can be retained by using single precision computation. The prospect proves economically more feasible as well, since the conservative ratio of prices for the GTX lineup versus professional GPUs is approximately 1/5.

Other additions to material complexity will increase the number of instructions issued, but much more importantly, increase the memory requirements in terms of thread-local storage and fetching. Additional needed memory stores and fetches will decrease performance well beyond the sole cost of

the execution of additional instructions, which is the case in damage computation. These additions would decrease performance but would cause more regular solution impacts, as opposed to somewhat erratic behavior exhibited currently. In the same way the performance impact of new material model descriptions (other hyperelastic, hypoelastic) will be predominantly in terms of new variables (e.g. damage or energy thresholds, history terms) that need to be stored and retrieved from GPU memory.

In general, texture memory usage did not elicit a significant impact on performance. The displacement variable is of high frequency usage and of no regular locality in memory, therefore textures were used to facilitate caching. However, clearly the three kernels perform a multitude of other operations such that the benefit of using texture fetching in one array is somewhat occluded. The improvement would likely be substantial in other applications that can exploit 2D or 3D data locality or interpolation which is intrinsically provided by texture fetching, as in e.g. image processing.

It is generally challenging to ascribe any specific design feature of a particular architecture directly to performance change. GPU subsystems are closely interconnected and interdependent. Nevertheless, it can certainly be said that the algorithm presented is primarily latency bound and its parallelization (and lack thereof) is memory driven. Very high register requirements needed to store the state variables (shape functions, displacements, stress, etc.) constrain the number of concurrent threads running, i.e. reduce device occupancy. The memory subsystem does compensate through caching but not without a speed penalty. Reducing the register requirements would be an excellent optimization as it would increase occupancy, but that proves very challenging for the algorithm at hand.

4.3. Future work

A robust FE implementation would account for element geometry and effective material response by changing the time-step and the coefficients accordingly, since the CFL condition should be satisfied throughout. Binary comparisons on a large set (e.g. all nodes or elements) and finding a smallest or largest value is a parallel reduction problem with efficient existing solutions (ref. [30]). It naturally follows that a variable-step central differences formula should be used. In the present case, through equations 15 and 2. In the testing cases of this paper, a static set of coefficients computed with a 0.8 weighting factor on initial stable time-step proved sufficient in our aim to ensure stability, not optimal convergence rates. Similarly, the convergence

rate coefficient C_r is constant throughout, but could be optimized during iteration to increase convergence time, as in an adaptive dynamic relaxation scheme (ref. [31]).

A reliable termination criterion is also essential for a robust and stand-alone FE code. The implementations of the time-step monitoring and reliable termination routines are left for future work. A termination criterion based on absolute error in the computed displacements presented in [23] is a potential candidate.

Future work also mandates the addition of an hourglassing preventive algorithm. This is crucial as almost any loading case aside from the possible uniform compression or extension is heavily affected by hourglass modes due to the under-integrated nature of used hexahedra. Encouragingly, existing anti-hourglassing solutions in a total Lagrangian setting (cfr. [32]) are pleasingly parallel. The implementation of other relevant anisotropic material models, dedicated to specific tissues like the aorta, are planned for future work, as well as different ancillary algorithms analogous to damage tracking.

5. Conclusion

We have presented an analysis of a CUDA implementation of the Total Lagrangian Explicit Dynamic algorithm, with potential of being used for training purposes or in an on-line surgical setting. The accuracy of the implementation has been compared to the industry-proven code Abaqus. Parallel solution speeds have been studied on a range of GPUs and reported in relation to a basic Abaqus/Explicit solution scheme. The impact of single and double precision, usage or non-usage of texture memory and the computational impact of using a damage model has been investigated.

The work presents a general image of the achievable solution speeds for nonlinear FE using the Total Lagrangian Explicit Dynamic algorithm with a simple material model and displacement loading on the GPU.

The results show significant speedups and good accuracy in comparison to stable and reliable solutions. This work, therefore, encourages the prospect of the described technology becoming implemented in the modern surgical theater or training systems in the near future.

Acknowledgment

This research was supported by an FWO research project (G.0.932.11.N.10). The authors would like to thank Prof. K.Miller and Dr. G.Joldes for instruc-

tion and assistance with the details of the TLED algorithm. We would also like to thank Sergio Portolés, Johan Kerkhofs, Phuong Toan Tran, Peter Vanden Berghe, Mickael Boland and Bart Van Elderen for general technical help and coding advice.

6. References

[1] J. Mosegaard, P. Herborg, T. S. Sørensen, A gpu accelerated spring mass system for surgical simulation, Studies in health technology and informatics.

[2] A. Rasmusson, J. Mosegaard, T. S. Sørensen, Exploring parallel algorithms for volumetric mass-spring-damper models in cuda, ISBMS '08 Proceedings of the 4th international symposium on Biomedical Simulation (2008) 49 – 58.

[3] U. Meier, O. Lopez, C. Monserrat, M. Juan, M. Alcaniz, Real-time deformable models for surgery simulation: a survey, Computer Methods and Programs in Biomedicine 77 (2004) 183–197.

[4] A. Liu, F. Tendick, K. Cleary, C. Kaufmann, A survey of surgical simulation: Applications, technology, and education, Presence 12 (2003) 599–614.

[5] M. BRO-NIELSEN, Finite element modeling in surgery simulation, Proceedings of the IEEE 86 (3) (1998) 490–503. doi:10.1109/5.662874.
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=662874

[6] N. A. Stephane Cotin, Herve Delingette, Real-time elastic deformations of soft tissues for surgery simulation, IEEE Transactions on Visualization and Computer Graphics 5 (1) (1999) 62–73. doi:10.1109/2945.764872.
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=764872

[7] G. Picinbono, H. Delingette, N. Ayache, Nonlinear and anisotropic elastic soft tissue models for medical simulation, in: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on, Vol. 2, 2001, pp. 1370–1375. doi:10.1109/ROBOT.2001.932801.

URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=932801

[8] G. Picinbono, H. Delingette, N. Ayache, Non-linear anisotropic elasticity for real-time surgery simulation, Graphical models 65 (2003) 305–321.

[9] J. Berkley, G. Turkiyyah, D. Berg, M. Ganter, S. Weghorst, Real-time finite element modeling for surgery simulation: an application to virtual suturing, IEEE Transactions on Visualization and Computer Graphics 10 (3) (2004) 314–325. doi:10.1109/TVCG.2004.1272730.
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1272730

[10] O. Comas, Z. A. Taylor, J. Allard, S. Ourselin, S. Cotin, J. Passenger, Efficient nonlinear fem for soft tissue modelling and its gpu implementation within the open source framework sofa, Lecture notes in computer science.

[11] E. D. Cris Cecka, Adrian J. Lew, Assembly of finite element methods on graphics processors, INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING 85 (2011) 640669.

[12] J. Wong, E. Kuhl, E. Darve, A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems, International Journal for Numerical Methods in Engineering (2015) n/a–n/adoi:10.1002/nme.4865.
URL http://dx.doi.org/10.1002/nme.4865

[13] K. Miller, G. Joldes, D. Lance, A. Wittek, Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation, Communications in Numerical Methods in Engineering on.

[14] G. R. Joldes, A. Wittek, K. Miller, Suite of finite element algorithms for accurate computation of soft tissue deformation for surgical simulation., Med Image Anal 13 (6) (2009) 912–919. doi:10.1016/j.media.2008.12.001.
URL http://dx.doi.org/10.1016/j.media.2008.12.001

[15] S. F. Johnsen, Z. A. Taylor, M. J. Clarkson, J. Hipwell, M. Modat, B. Eiben, L. Han, Y. Hu, T. Mertzanidou, D. J. Hawkes, S. Ourselin,

Niftysim: A gpu-based nonlinear finite element package for simulation of soft tissue biomechanics, International Journal of Computer Assisted Radiology and Surgery.

[16] V. Volkov, J. W. Demmel, Benchmarking gpus to tune dense linear algebra, SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing Article No. 31.

[17] Nvidia, CUDA C Programming Guide v7.0, Nvidia Corporation (2015). URL http://developer.nvidia.com/nvidia-gpu-computing-documentation

[18] D. Göddeke, GPGPU–Basic math tutorial, Tech. rep., Fachbereich Mathematik, Universität Dortmund, Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300 (Nov. 2005).

[19] A. Munshi (Ed.), The OpenCL Specification, 2012. URL https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[20] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113. URL http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x

[21] Nvidia, CUDA C Best Practices Guide v7.0, Nvidia Corporation (2015). URL http://developer.nvidia.com/nvidia-gpu-computing-documentation

[22] T. Belytchko, T. J. Hughes (Eds.), Computational methods for transient analysis, 1983.

[23] G. R. Joldes, A. Wittek, K. Miller, Computation of intra-operative brain shift using dynamic relaxation., Comput Methods Appl Mech Eng 198 (41) (2009) 3313–3320. doi:10.1016/j.cma.2009.06.012. URL http://dx.doi.org/10.1016/j.cma.2009.06.012

[24] T. Belytschko, A survey of numerical methods and computer programs for dynamic structural analy, Nuclear 31 (1976) 23–34.

[25] G. R. Joldes, A. Wittek, K. Miller, Real-time nonlinear finite element computations on gpu - application to neurosurgical simulation., Comput

Methods Appl Mech Eng 199 (49-52) (2010) 3305–3314. doi:10.1016/j.cma.2010.06.037.
URL http://dx.doi.org/10.1016/j.cma.2010.06.037

[26] Nvidia geforce gtx 980, Tech. rep., NVIDIA Corporation (2014).
URL http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[27] Kepler gk110 whitepaper, Tech. rep., NVIDIA Corporation (2009).
URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[28] Fermi whitepaper, Tech. rep., NVIDIA Corporation (2012).
URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[29] Dynamic parallelism in cuda, Tech. rep., Nvidia Corporation (2012).
URL http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf

[30] M. Harris, Optimizing parallel reduction in cuda, NVIDIA Corporation, 2013.
URL http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[31] G. R. Joldes, A. Wittek, K. Miller, An adaptive dynamic relaxation method for solving nonlinear finite element problems. application to brain shift estimation., Int j numer method biomed eng 27 (2) (2011) 173–185. doi:10.1002/cnm.1407.
URL http://dx.doi.org/10.1002/cnm.1407

[32] G. R. Joldes, A. Wittek, K. Miller, An efficient hourglass control implementation for the uniform strain hexahedron using the total lagrangian formulation, Communications in Numerical Methods in Engineering 24 (2008) 1315–1323.

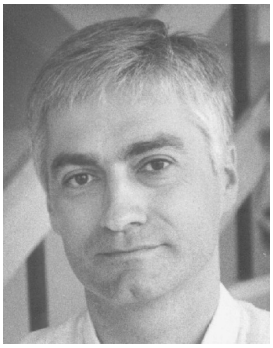
7. Vitae

7.1. Vukasin Strbac

Vukasin Strbac, MSc. (1985) received his MSc degree in Information Science from the Faculty of Organization and Informatics, University of Zagreb, Croatia, in 2009, majoring in low-level programming of computer graphics and rigid body dynamics. Professional experiences include graphics software development in TV broadcasting with NTH AG, until the start of his PhD program. From 2011 he is a PhD student at KU Leuven, Biomechanics section, department of Mechanical engineering. His interests are: parallel programming, many-core architectures, optimization and nonlinear solvers applied to finite element modeling. A presenter at GPU Technology Conference and currently working on efficient implementations of complex anisotropic models of arterial tissue.

7.2. Nele Famaey

Dr. Ir. Nele Famaey (1984) received her MSc and PhD degree in mechanical engineering from KU Leuven, Belgium, in 2006 and 2012, respectively. From 2012 on, she is a postdoctoral research fellow of the Flemish Research Fund (FWO) at the Department of Mechanical Engineering, KU Leuven. From 2010 to 2011, she was a Fulbright visiting PhD scholar at the Computational biomechanics lab of Stanford University. Her research interests include nonlinear finite element modeling of biological soft tissue, experimental characterization of material properties of cardiovascular tissue, damage detection and quantification in biological tissue, and computational aspects of robotic surgery, on which she authored over 20 publications.



7.3. Jos Vander Sloten

Prof. Dr. Ir. Jos Vander Sloten (1962) obtained his MSc and PhD degree in mechanical engineering from KU Leuven in 1985 and 1990, respectively. He is currently a full professor and former chair of the Biomechanics Section at KU Leuven. He also chairs the Leuven Medical Technology Centre (L-MTC). His teaching assignments are engineering mechanics, problem solving and engineering design, computer integrated surgery systems. His research interests are computer applications in musculoskeletal biomechanics and computer integrated surgery, on which he authored more than 160 journal papers. He is member of the council of the Belgian Society for Medical and Biological Engineering and Computing, and a former council member of the European Society of Biomechanics. In the European Alliance for Medical and Biological Engineering and Science (EAMBES) he served as secretary-general (2003-2004), president-elect (2005) and president (2006). He was elected Founding Fellow of EAMBES. He is a co-founder of the spin-off company Custom8, member of the board of directors of the company Materialise NV and provides consultancy to the company Mobelife NV.